Today, the third and final challenge from 0xgiraffe was released. Are we gonna do it? Of course we are!

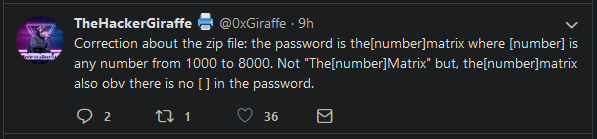






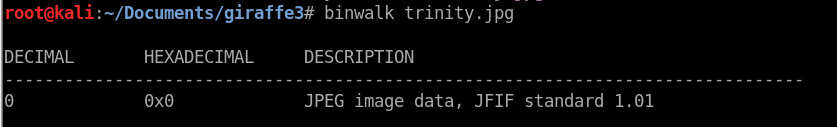


Looks pretty similar to all the other challenges so far. Download a file, get an IP from the file, etc. A few more hints show up later, reading:

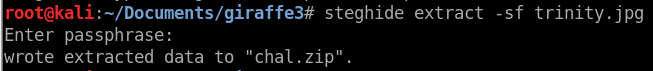




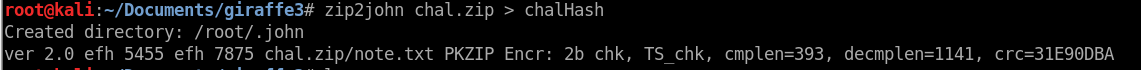
Let’s get started. The picture you download from the link in the tweet is this picture of Trinity from the movie The Matrix. We already got some strong hints on what we need to do with this image, but let’s try some of our other tricks first just to see what happens. Checking the metadata for hidden files, we use **binwalk** on the image like so:



Nothing to see here, just a normal JPEG image. Let’s try the **steghide** command. Check the manual using **man steghide** if you need more syntax, but we will start with the most basic form:



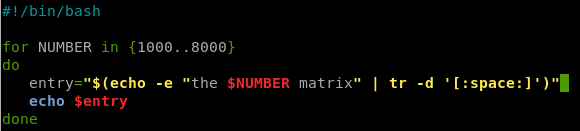
The **extract** line specifies that we are extracting data, not hiding it, and the **-sf** specifies the target file. One of our hints tells us to use a blank password, so we do that. We have successfully extracted the zipped file **chal.zip** from inside the image using steganography, pretty cool. We have another hint about the zip file, namely that the password is the(some number)matrix , with the number between 1000 and 8000. There are a few ways to automate the process of cracking this, but we will do a dictionary attack against the file. The first thing we need to do is extract the password hash from the zip file:



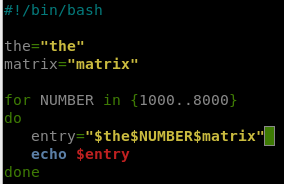
The **zip2john** command extracts hash files from a password-encoded zip file. We saved the hash to **chalHash**, which looks like this:



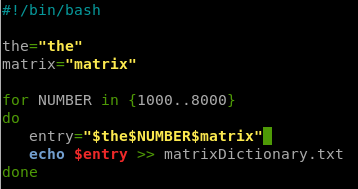
Next, we need to create a dictionary to use for the cracking. I’m just going to create a quick script to automatically generate these.



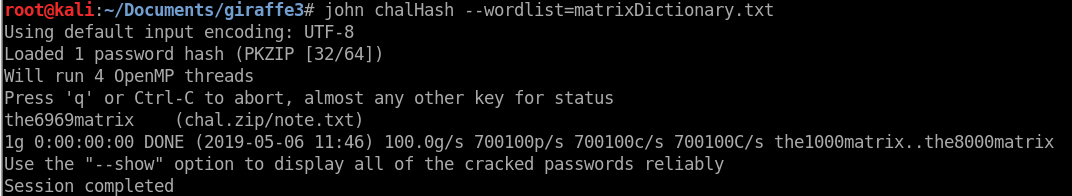
This creates a basic for loop, going from 1000 to 8000. It basically creates a variable **entry** that is a string consisting of “the $NUMBER matrix”, where $NUMBER is substituted for the current value of the loop. The second half uses the **tr** command to trim the whitespace, since putting the two strings right next to the $NUMBER messes up the syntax. This is actually a terrible way to implement this, so let’s improve it:

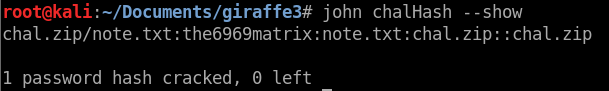


This method is actually much more efficient, as it doesn’t require those pesky trim operations, and finishes execution almost instantaneously instead of taking ten seconds. The main difference is that we instantiated the other strings in the password, the and matrix, as variables so they can be easily added together without spaces. We need to make one final modification, we must make it add each entry to a file, rather than simple echoing it:

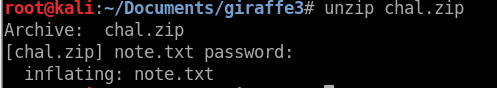


Make sure you use **>>** to write the file instead of **>**. A single greater-than sign will overwrite the existing file with the input, while double signs will simply add the input to the end of the file. If you cat the new file **matrixDictionary.txt** or open it in an editor, you will see that it is a nice, clean file containing all of the possible passwords. Now we can use **John the Ripper** to finish the job:



Now we use the **–show** tag to view our results. 

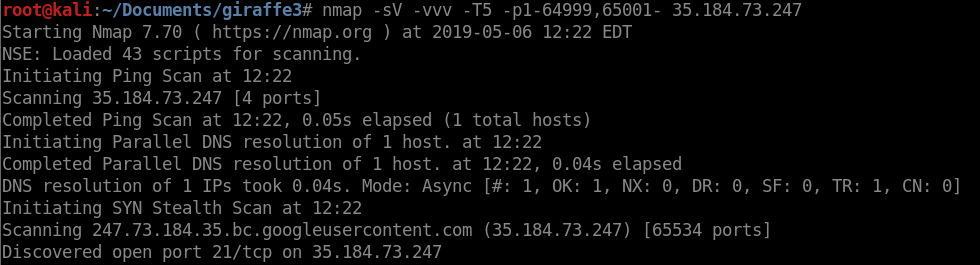
Looks like the password is **the6969matrix**. Wow. Anyways, let’s unzip the file with the password.



Let’s see what the **note.txt** has to say:



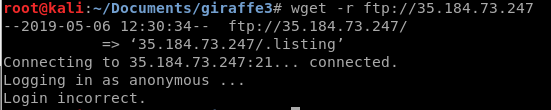
We’ve got our IP! You know what comes next! Scanning! Let’s use **nmap** to find a port we can work on. We’ll use **-sV** for version enumeration on open ports, **-vvv** for verbose output (don’t need to wait for the scan to finish to see found ports), **-T5** to speed things up a bit, and **-p1-64999,65001-** to scan all ports except for the banned port 65000.



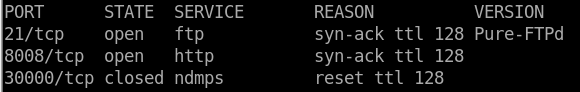
We’ve found an FTP port! Well, I assume its FTP because it is port 21, which tends to be that, but whatever. The scan is still going, but until we find other ports we can play with this one in the meantime. Let’s see if we can connect.



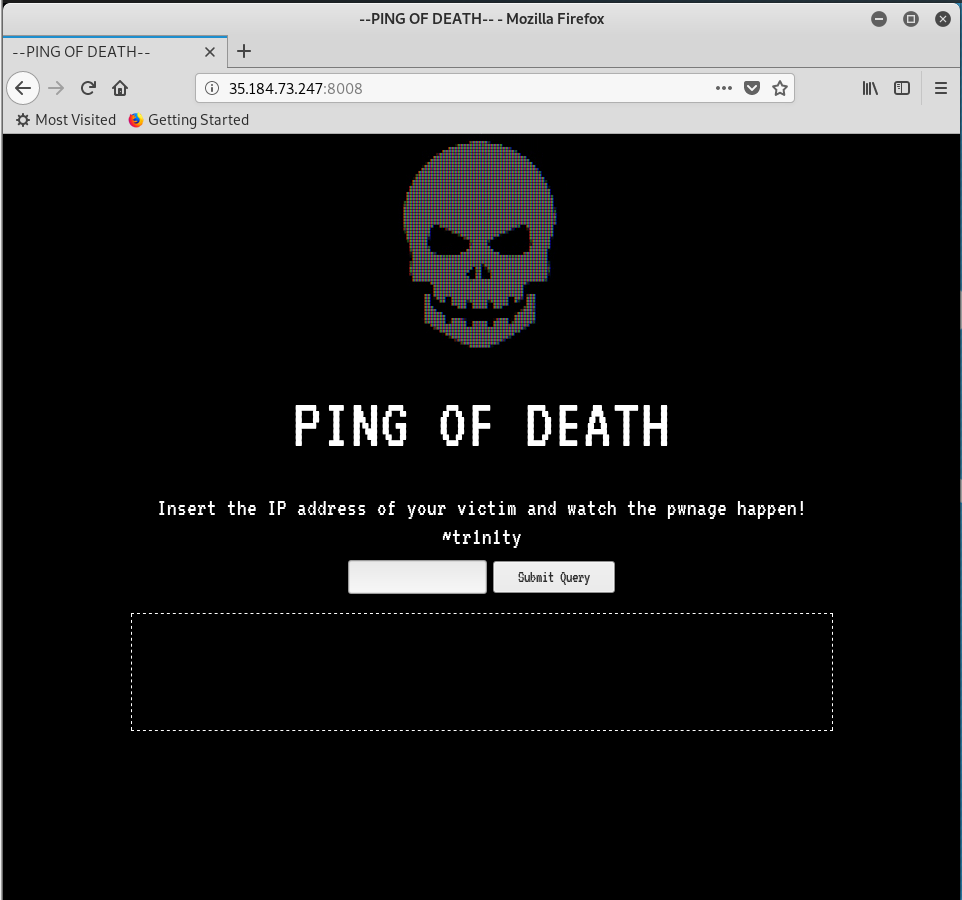
Hmm… Can’t connect with FTP, but before we give up let’s try a different command. The **wget** command is great for pulling web content from pretty much anywhere on the internet, and supports many protocols such as http and ftp. Let’s see if we can grab the content.



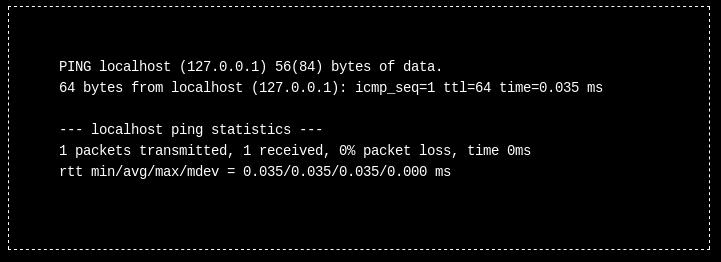
That didn’t work either, so we will have to try a bit harder. While we were messing around, the scanner finished and found another port:



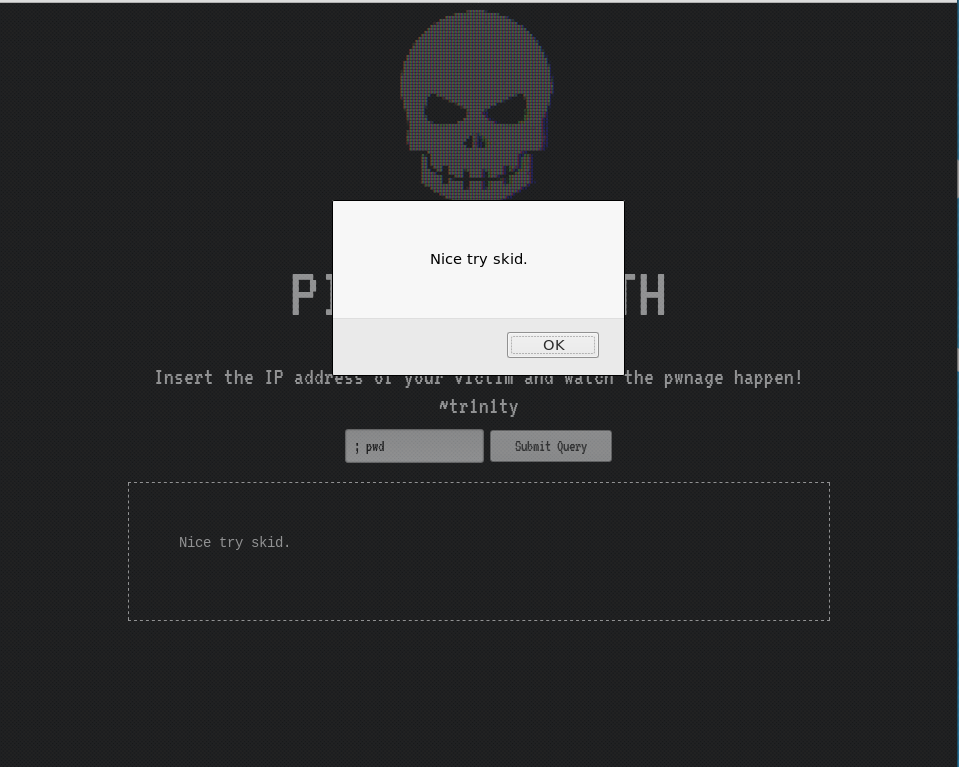
After the port 30000 is a long list of similar closed ports, so I omitted them. However, not only did we find an http service on port 8008, but we also found that our ftp port is running Pure-FTPd service. This could help us figure out how to connect later, but we will focus on the web thingy for now. Let’s try to navigate to port 8008 in a browser.



So we’ve found the ping of death! I wonder what it actually does… It has a form for a query and an empty box, so let’s see what happens if we make the service “attack” itself! This can be done three ways: using it IP we have (35.184.73.247), using 127.0.0.1 (the loopback address every device uses to refer to itself) or just **localhost**. Let’s try that.



I entered just **localhost** in the box, and this is what we got. As you may recognize, this is literally just the output of a ping command. This means we can probably get **remote command injection** from the service, making it give us the output of any command. The most basic form of this is putting a semicolon and another command like so:



Okay, so the page is protected from the simplest form of this attack, but how is it doing this? If it is simply detecting the semicolon, we can use the escape character representing a semicolon to do the same attack. Let’s enter nothing but a semicolon to check if it gives the same error:



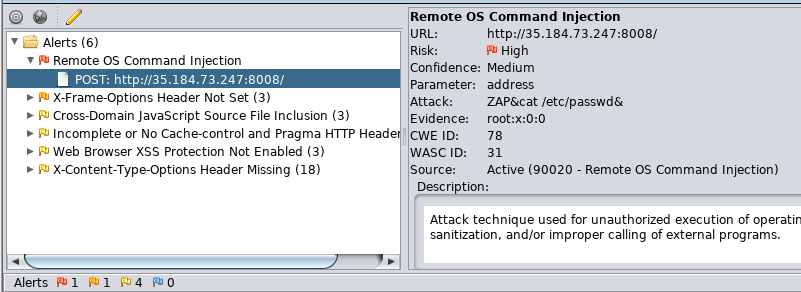
Yep, it does. Let’s try it with the bash escape character. Bash escape characters are literally just a **%** symbol followed by the hexadecimal value that represents the character. The hex for a semicolon is **3B**, so let’s try **%3B** as an input**.**



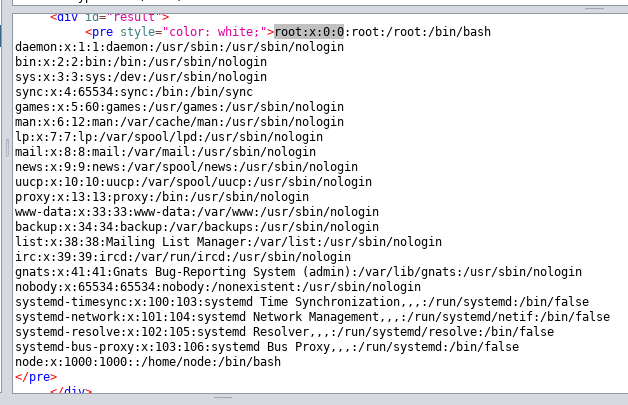
That didn’t work either… Okay, let’s try something different. Let’s use OWASP Zap to see if they can find proper command injection syntax: Well, first let’s admire the sweet ascii message in the page’s source:



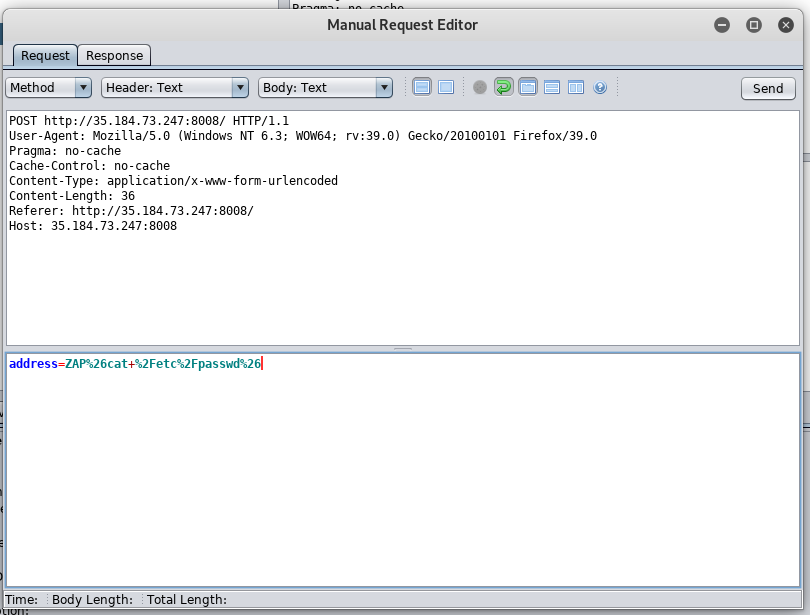
Awesome. Nice Matrix quote too. Now, back to business:



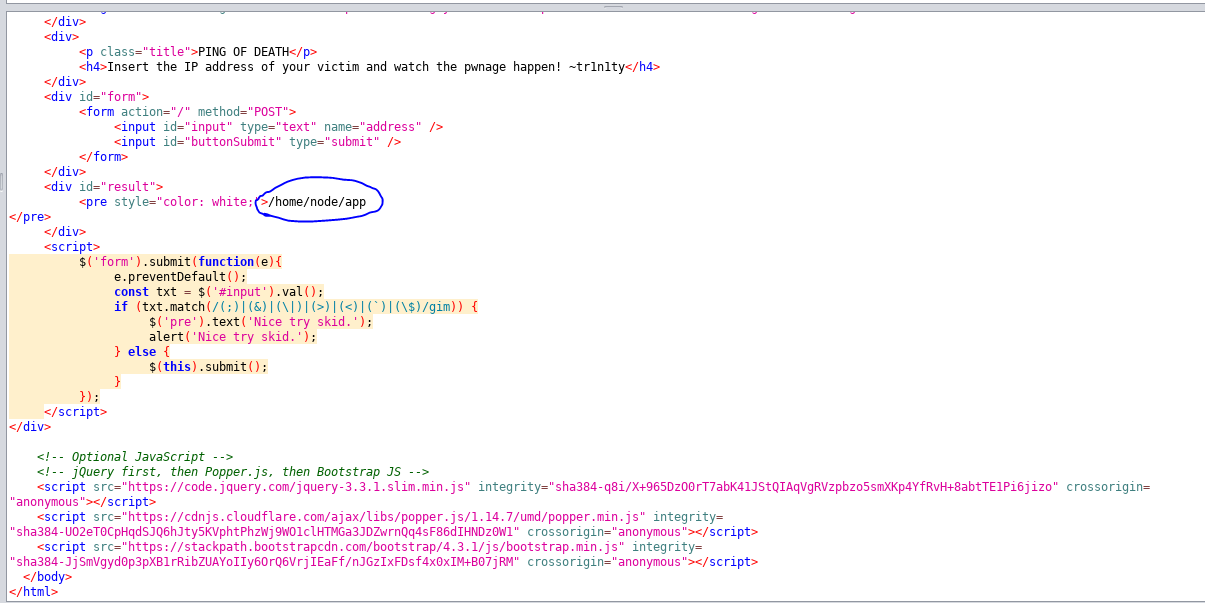
ZAP was able to find an injection technique pretty fast! It got a return on the /etc/passwd file in the page response:



Even though the script was still tripped, the proper answer was still contained in the HTTP response, so we can read it using proxies such as ZAP. By resending this request, we can look around for other interesting stuff on the machine.

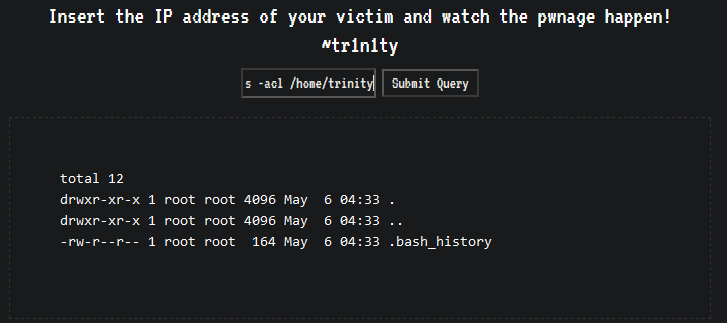


Looking at the request, we can see the default command injection uses the same strategy we tried earlier, except they used the escape character for **&** as well as **/**, so let’s resend this packet and see what we can see. Interestingly enough, it looks like our entry still triggers the anti-injection script, but the command output is still in the response itself… Neat. This time, we injected the **pwd** command:

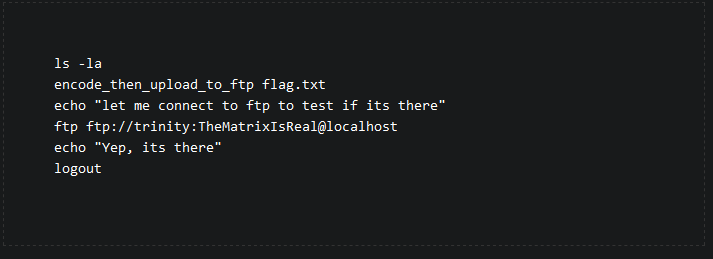


Looks like the page loads from a **nodejs** app. Let’s see what we can find. Going back to our hints, we can see that we are supposed to look under the Trinity user and see what is there, no reverse shells. That is nice, because reverse shells are a pain in the butt sometimes!

Okay, before we do that, there is actually an easier way to do this. We can just use the **NoScript** browser extension to disable the script that stops **RCE**. So let’s do that. **;ls -acl /home/trinity**



Now we don’t need to worry about escape characters. The only thing here is the .bash\_history, so let’s look at that. **;cat /home/trinity/.bash\_history**



Wow, we pretty much just got nice directions for how to connect to the ftp server, yay! Let’s do that now. Rather than using the ftp command, we will just use wget as follows:



The **-r** command recursively downloads, in case there are directories. The rest specifies the username, password, and target ftp server. Our result is a file called **flag.encoded**. Here is the contents:

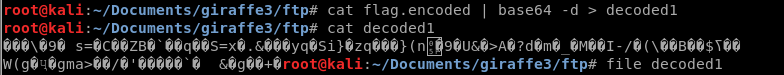


It is clearly encoded in some way, as itsextension seems to indicate. Let’s try base64 decoding it:



That didn’t work… We can tell since there are invalid characters that this isn’t another case of base64 within base64, so let’s try a different encoding… actually, let’s see if we can figure out what the flag’s encoding is on the server. We see above that there is a file or alias called **encode\_then\_upload\_to\_ftp** in some directory. Assuming that file still exists, if we find it and read it, we can see how the flag was encoded in order to decode it. Let’s do that now:

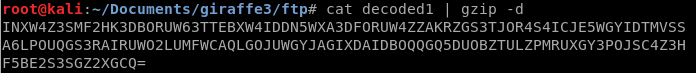
Actually, after fiddling around for a while, I realized that the base64 decode was the correct option after all... Here’s how you know. Even though the data we got from the screenshot above is nonsense, if we save it to a file we can analyze it further:



We **base64 -d** decode it into a file simply called **decoded1.** Let’s use the **file** command on it:



It’s **gzip** compressed data? We know how to handle that!



Nice, that looks like a base64 or base32 string. Let’s save it to a file called **decoded2** and decode it again!



It was base32, and now we have our flag! Fantastic! We did it! Yay etc.